



BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY



Leveraging Compiler-Based Tools for Performance Portability

**Protonu Basu⁺, Samuel Williams⁺, Brian Van
Straalen⁺, Leonid Oliker⁺, Philip Colella⁺, Mary Hall^{*}**

⁺ Lawrence Berkeley National Laboratory

^{*} University of Utah

Performance and Productivity Challenge – GSRB Smooth

```
/* Laplacian 7-point Variable-Coefficient Stencil */
```

```
for (k=0; k<N; k++)
```

```
for (j=0; j<N; j++)
```

```
for (i=0; i<N; i++)
```

```
temp[k][j][i] = b * h2inv * (
    beta_i[k][j][i+1] * ( phi[k][j][i+1] - phi[k][j][i] )
    +beta_i[k][j][i] * ( phi[k][j][i] - phi[k][j][i-1] )
    +beta_j[k][j+1][i] * ( phi[k][j+1][i] - phi[k][j][i] )
    -beta_j[k][j][i] * ( phi[k][j][i] - phi[k][j-1][i] )
    +beta_k[k+1][j][i] * ( phi[k+1][j][i] - phi[k][j][i] )
    -beta_k[k][j][i] * ( phi[k][j][i] - phi[k-1][j][i] ) );
```

```
/* Helmholtz */
```

```
for (k=0; k<N; k++)
```

```
for (j=0; j<N; j++)
```

```
for (i=0; i<N; i++)
```

```
temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] -
    temp[k][j][i];
```

```
/* Gauss-Seidel Red Black Update */
```

```
for (k=0; k<N; k++)
```

```
for (j=0; j<N; j++)
```

```
for (i=0; i<N; i++){
```

```

if ((i+j+k+color)%2 == 0 )
    phi[k][j][i] = phi[k][j][i] - lambda[k][j][i] *
    np[k][j][i] - rhs[k][j][i]);}

```

[illegible]

Code A: miniGMG baseline smooth operator approximately 13 lines of code

Code B: miniGMG optimized smooth operator approximately 170 lines of code

GPU code for GSRB Smooth

[illegible][illegible][illegible]

Code C: miniGMG optimized smooth operator for GPU, 308 lines of code for just kernel

Background: Challenges

- Performance portability

Across fundamentally different CPU and GPU architectures

- Programmer productivity

High performance implementations will require low-level specification in standard MPI+OpenMP, CUDA

- Software maintainability and portability

May require maintaining multiple implementation of same computation

Possible ways to address the challenges

- Follow MPI and OpenMP standards

Same code unlikely to perform well across CPU and GPU

Low level specification may be required for high-performing OpenMP

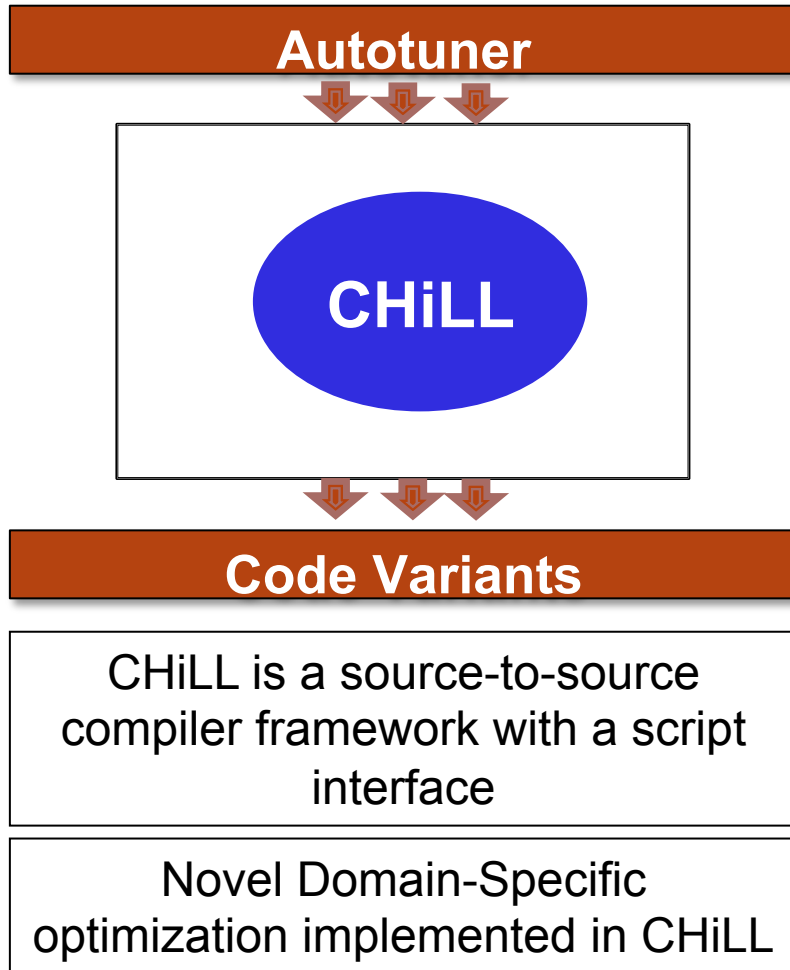
Vendor C and Fortran compilers not optimized for HPC workloads

- Some domain-specific framework strategies

Libraries, C++ template expansion, standalone DSL

Not **composable** with other optimizations

Compiler Based Approach



- Exploit existing compiler transformations to accomplish optimization goals
- Develop new domain-specific transformations and required analysis and code generation support
 - Supports autotuning

Compiler Based Approach

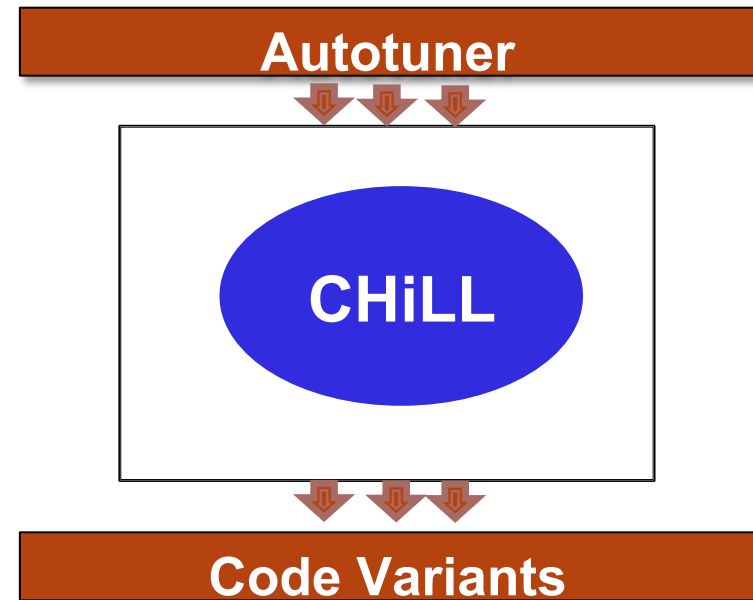
- Composable transformation and code generation

Leverage rich set of existing transformations and code generation capability

Mathematically represented using polyhedral framework

- Extensible to new domain-specific transformations and decision algorithms

Compose with existing transformations

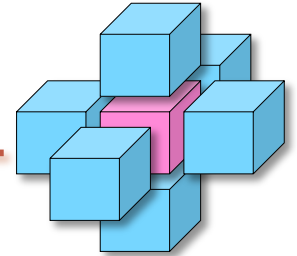


Experience with CHiLL

	Input	Existing Transformations	Domain-specific transformations	Autotuning	
Geometric Multigrid	Sequential C computation (w/ MPI and OpenMP harness)	Communication-avoiding: fusion, tile, wavefront (skew&permute), OpenMP, CUDA	Ghost zones, Partial sums	Ghost zone depth, threading, strategy at each level of V-cycle	
Tensor Contraction	Mathematical Formula	Tile, permute, scalar replacement, unroll, CUDA	Rewriting, Decision algorithm	Loop order, CUDA threading	
Sparse Matrix Computation	Sequential C with CSR matrix	Tile, permute, skew, unroll, reduction, scalar expansion, OpenMP, CUDA	Generate inspectors, coalesce, make-dense, compact, split, level sets	Threading, matrix repr.	

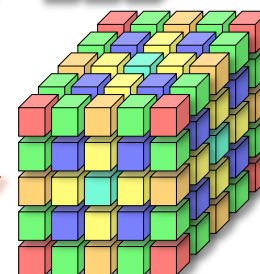
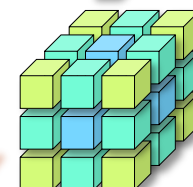
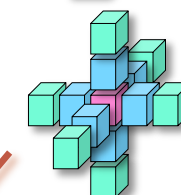
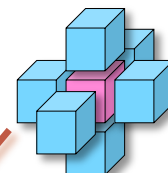
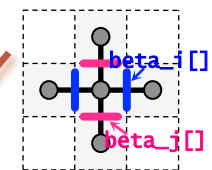
Performance Bottlenecks

Stenc	Coefficie	Iteratio	Flop	Byte	AI
7-point	Constant	Jacobi	8	24	0.33



Performance Limited by
Memory Bandwidth!

Stencil	Coefficient	Iteration	Flop	Byte	AI
7-point	Variable	GSRB	17	80	0.21
		Jacobi	17	48	0.35
7-point	Constant	Jacobi	8	24	0.33
13-point	Constant	Jacobi	15	24	0.63
27-point	Constant	Jacobi	32	24	1.33
125-point	Constant	Jacobi	134	24	5.58

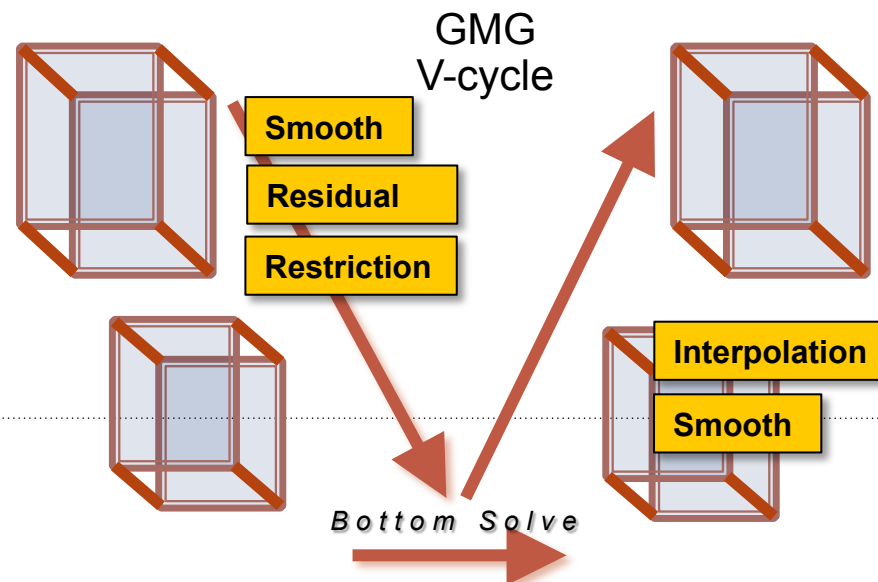
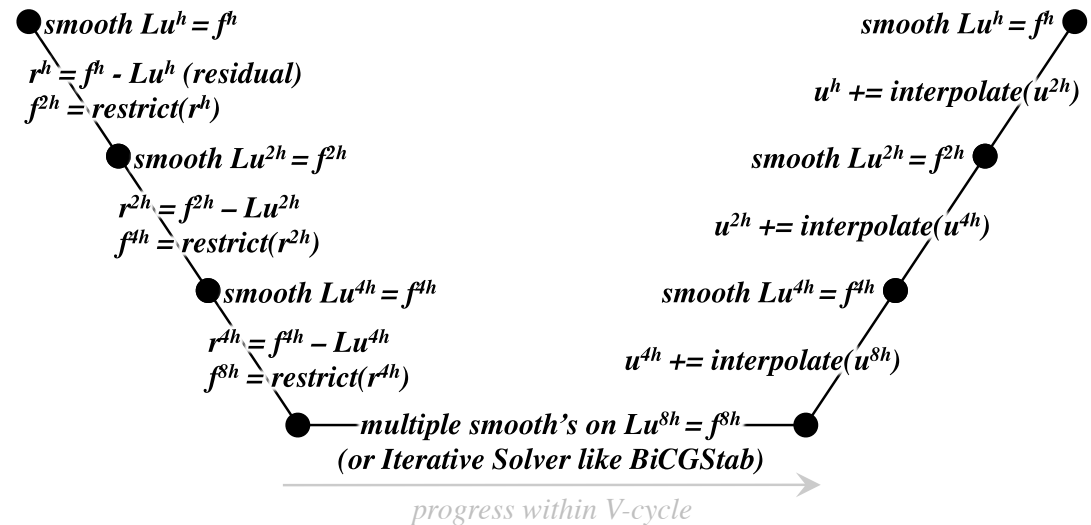


Increasing Flops/Byte

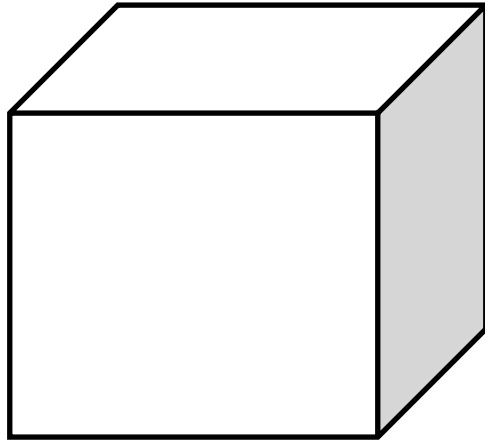
Geometric Multigrid (GMG)

MG is a hierarchical approach to solving the linear system $Ax=B$

GMG solves the linear system $Ax=B$, where A is a stencil applied on a grid

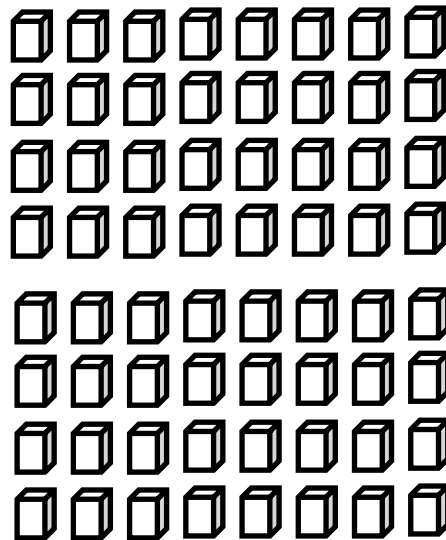


Domain 256^3

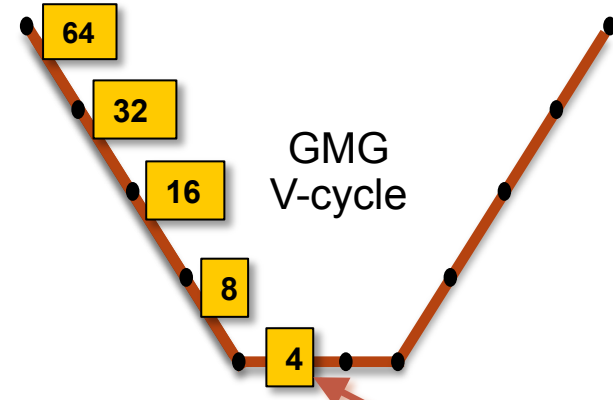


miniGMG

List of 64^3 Boxes
Computed In Parallel (OMP)

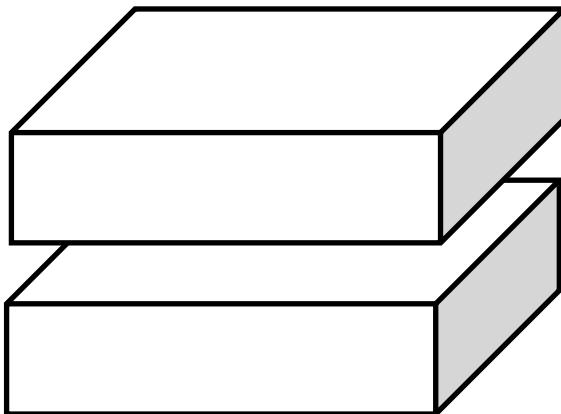


4 iterations
of smooth



48 iterations
of Smooth

Domain
decomposed to MPI
processes (2)



**Smooth Dominates
Runtime**

Stencil/Smooth

CA

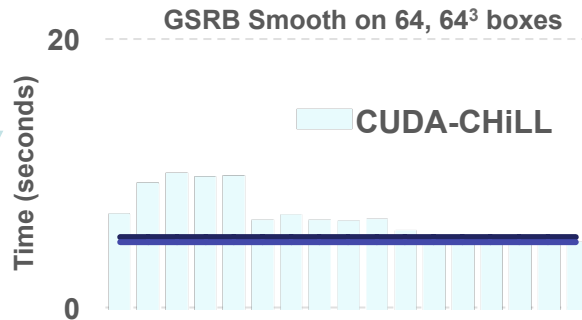
Communication-Avoiding Optimizations

PS

Stencil Reordering: Partial Sums

Memory Bandwidth Bound

CA



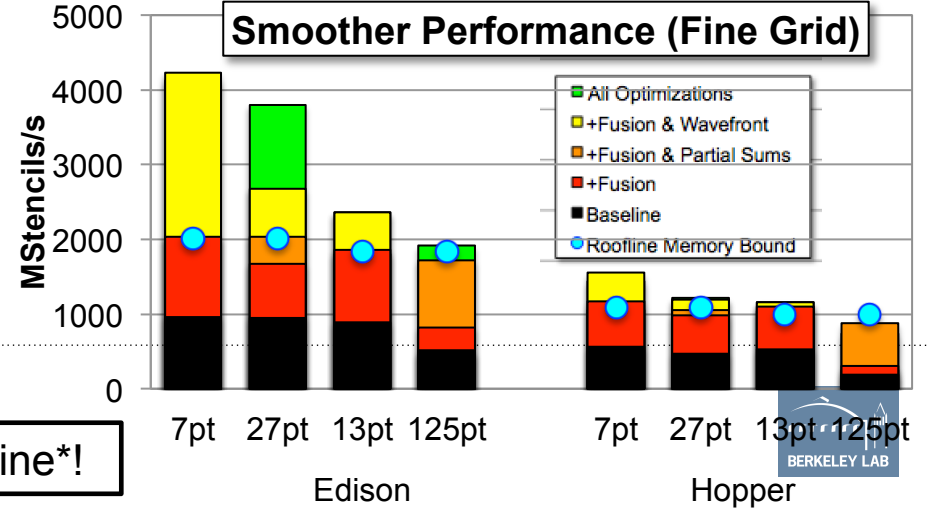
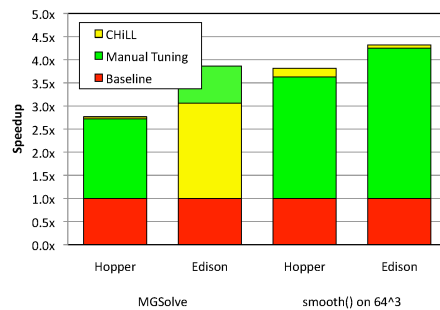
PS

Compute Bound by High FLOP intensity and Poor Register Reuse

CA

Memory Bandwidth Bound

Compiler Autotuning Matches Manual Tuning!



Compiler Autotuning Beats Roofline*!

Baseline GSRB Smooth

S0

```
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      /* statement S0 */
      temp[k][j][i] = b * h2inv * (
        beta_i[k][j][i+1] * ( phi[k][j][i+1] - phi[k][j][i] )
        -beta_i[k][j][i] * ( phi[k][j][i] - phi[k][j][i-1] )
        +beta_j[k][j+1][i] * ( phi[k][j+1][i] - phi[k][j][i] )
        -beta_j[k][j][i] * ( phi[k][j][i] - phi[k][j-1][i] )
        +beta_k[k+1][j][i] * ( phi[k+1][j][i] - phi[k][j][i] )
        -beta_k[k][j][i] * ( phi[k][j][i] - phi[k-1][j][i] ) );
```

7point VC
stencil

S1

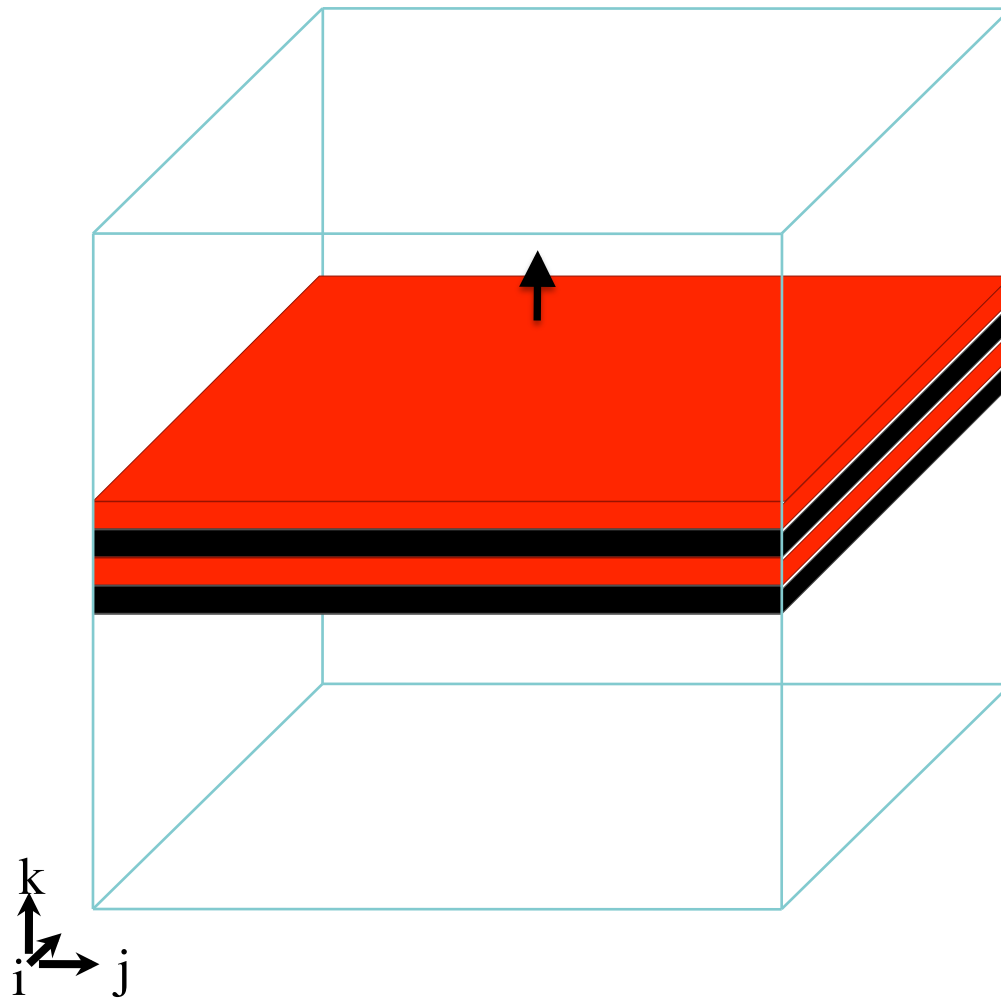
```
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      /* statement S1 */
      temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] - temp[k][j][i];
```

GSRB
update

S2

```
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++){
      if ((i+j+k+color)%2 == 0 )
        /* statement S2 */
        phi[k][j][i] = phi[k][j][i] - lambda[k][j][i] *(temp[k][j][i] - rhs[k][j][i]);}
```

Wavefront: Reducing Vertical Communication



Wavefront fuses multiple grid sweeps reducing DRAM traffic

Wavefront: Reducing Vertical Communication



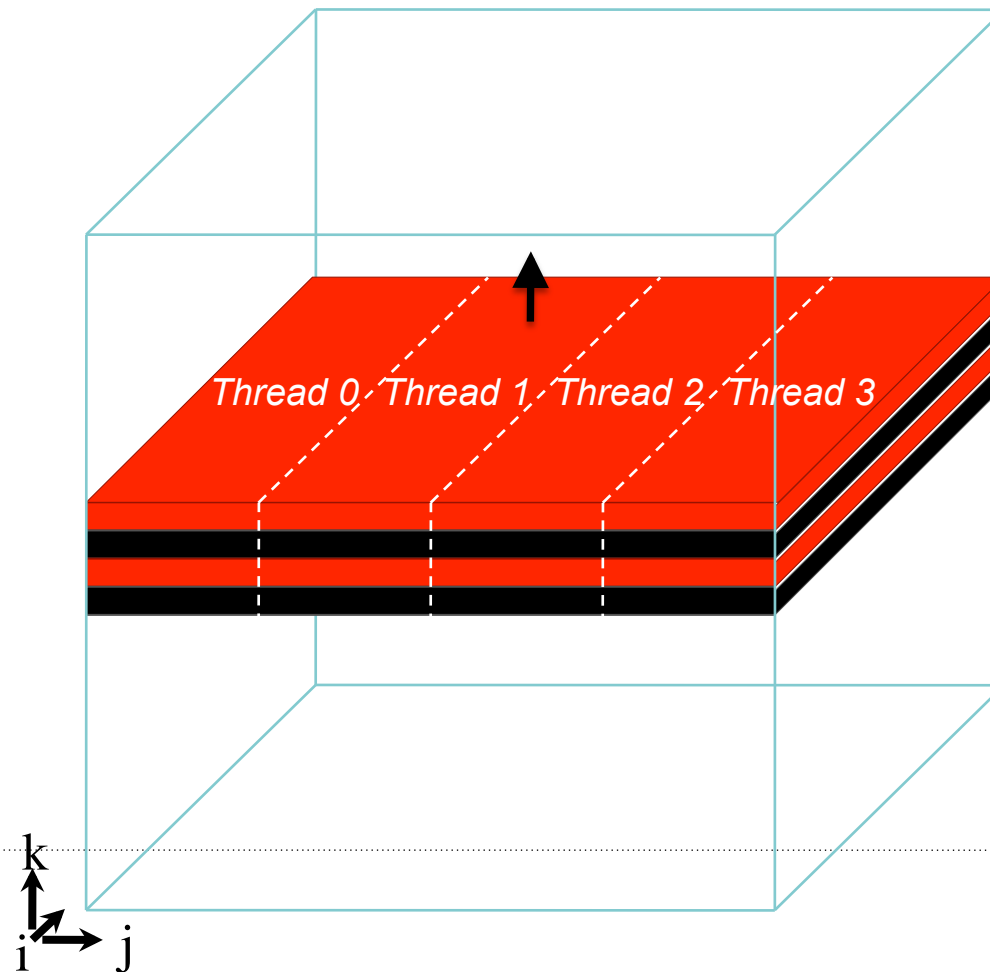
Wavefront = Loop Skew + Loop Permute

We tune to find the ghost zone depth and wavefront depth!

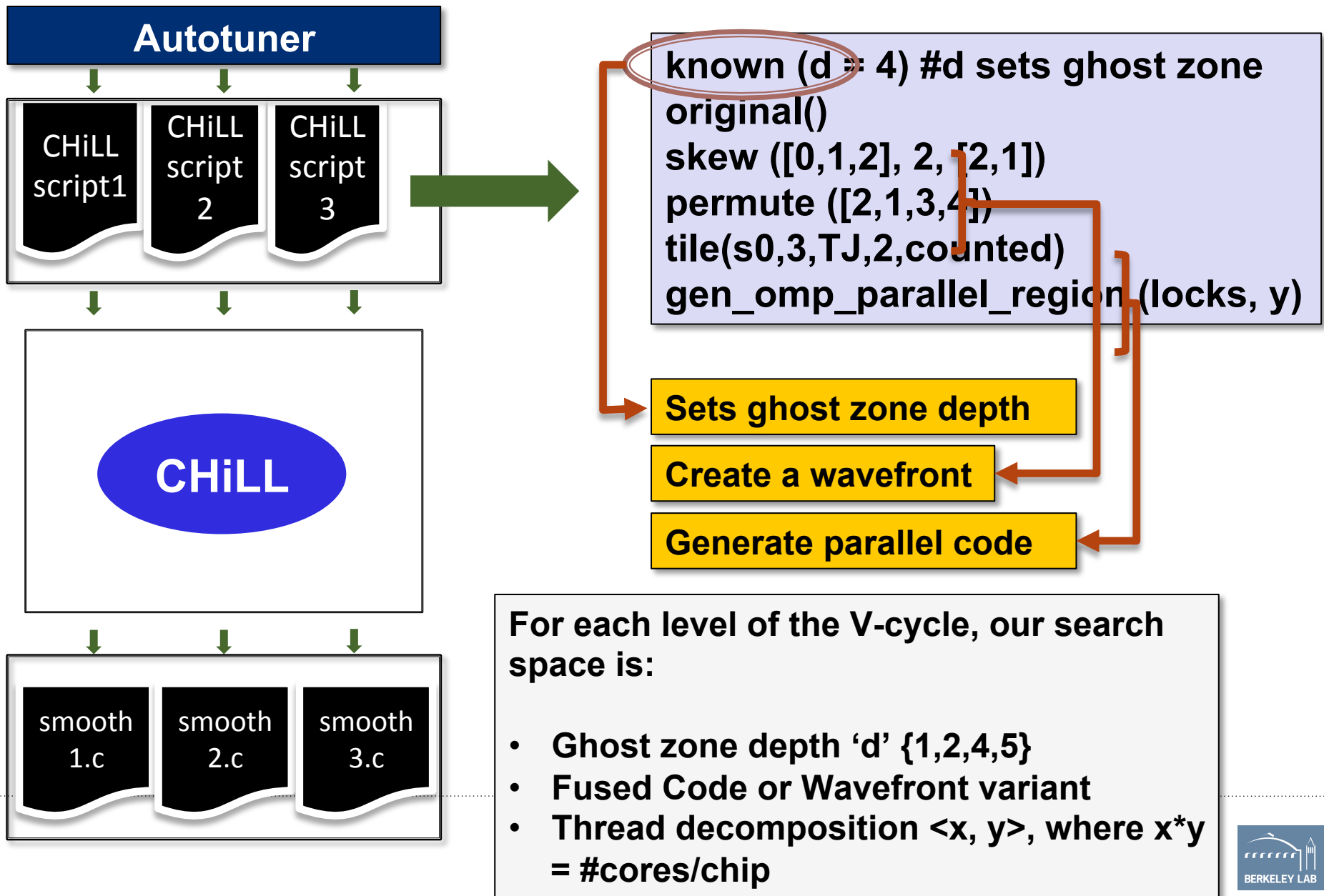
OpenMP Code Generation: Nested Parallelism

Wavefront has a larger working set

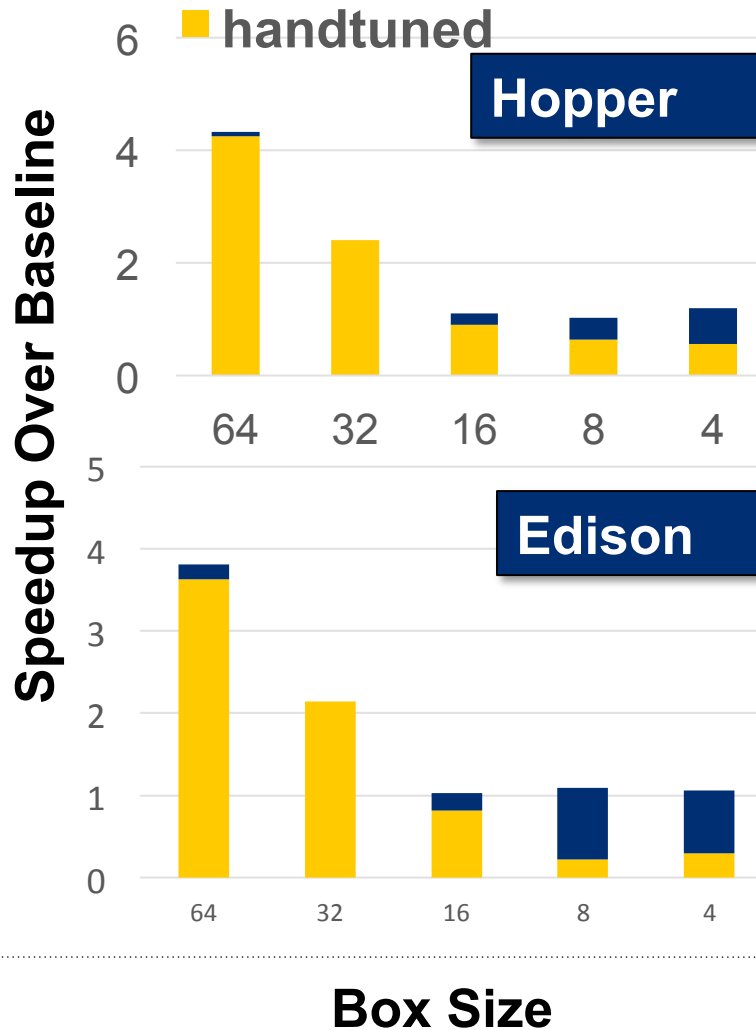
Thread blocking to manage working set



Experimental Methodology



Performance of GSRB Smooth

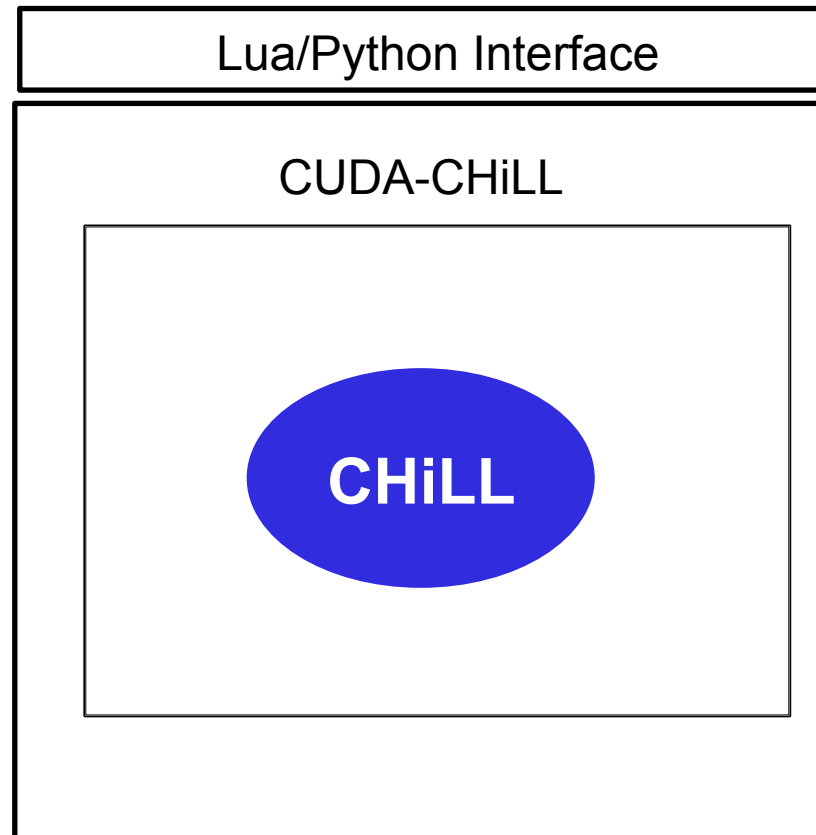


Box	Ghost Zone Edi/Hop	Thread Decomposition <outer, inner>		Code Variant Edi/Hop
		Edison	Hopper	
64	4	<4,3>	<2,3>	Wave
32	4	<4,3>	<2,3>	Wave
16	2	<12,1>	<6,1>	Wave
8	2	<12,1>	<6,1>	Fused
4	2	<12,1>	<6,1>	Fused

Manual tuning spent considerable effort on finer 64^3 boxes but did not specialize for smaller boxes

Autotuning picked nested- parallelism for finer boxes; manually tuned code used intra-box threading

CUDA-CHiLL



CUDA-CHiLL is a thin layer built on top of CHiLL to generate CUDA code

Deconstructs (tiles) a loop nest, and assigns loops to threads and blocks

Parallelization via Loop Tiling

Input GSRB smooth

```
for(box=0; box<64; box++){
  for(k=1; k<=64; k++){
    for(j=1; j<=64; j++){
      for(i=1; i<=64; i++){
        if(( i+ j + k + (color) ) % 2 == 1 ) {
          S0();
          S1();
          S2();}}}}}
```

BZ is fixed to 64 (number of boxes)

Tune to find best value of TX, TY
(dimensions of 2D block)

BX=64(box size)/TX, BY=64(box size)/TY

Tiled loop nest with loops marked for blocks/threads

mark as block dim z (BZ=64)

mark as block dim y (BY=4)

mark as thread dim y
(TY=16)

mark as block dim x (BX=2)

mark as thread dim x (TX=32)

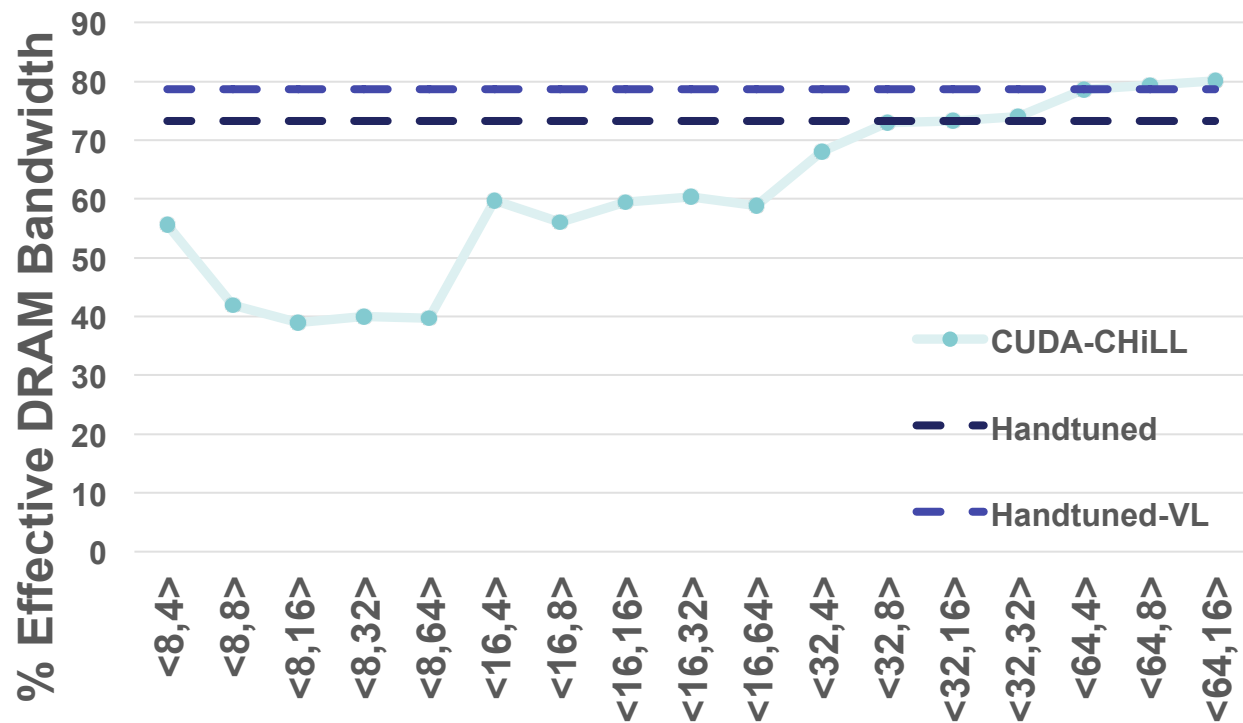
```
for(box = 0; box <= 63; box++) {
  for(k = 1; k <= 64; k++) {
    for(jj = 0; jj <= 3; jj++) {
      for(j = 0; j <= 15; j++) {
        for(ii = 0; ii <= 1; ii++) {
          for(i = intMod(-j-k-color-1,2); i <= 31; i += 2) {
            S0();
            S1();
            S2(); }}}}}}
```

CUDA-CHILL

```
/* gsrb.lua, variable coefficient GSRB, 643 box size */  
init("gsrb_mod.cu", "gsrb",0,0)  
dofile("cudaize.lua") # custom commands in lua  
  
# set up parallel decomposition, adjust via  
autotuning  
TI=32  
TJ=4  
TK=64  
TZ=64  
  
tile_by_index(0, {"box","k","j", "i"},{TZ,TK, TJ, TI},  
{l1_control="bb", l2_control="kk", l3_control="jj",  
l4_control="ii"},{"bb","box","kk","k","jj","j","ii","i"})  
  
cudaize(0, "kernel_GPU",  
{_temp=N*N*N*N,_beta_i=N*N*N*N,  
_phi=N*N*N*N},{block={"ii","jj","box"},  
thread={"i","j"}},{})
```

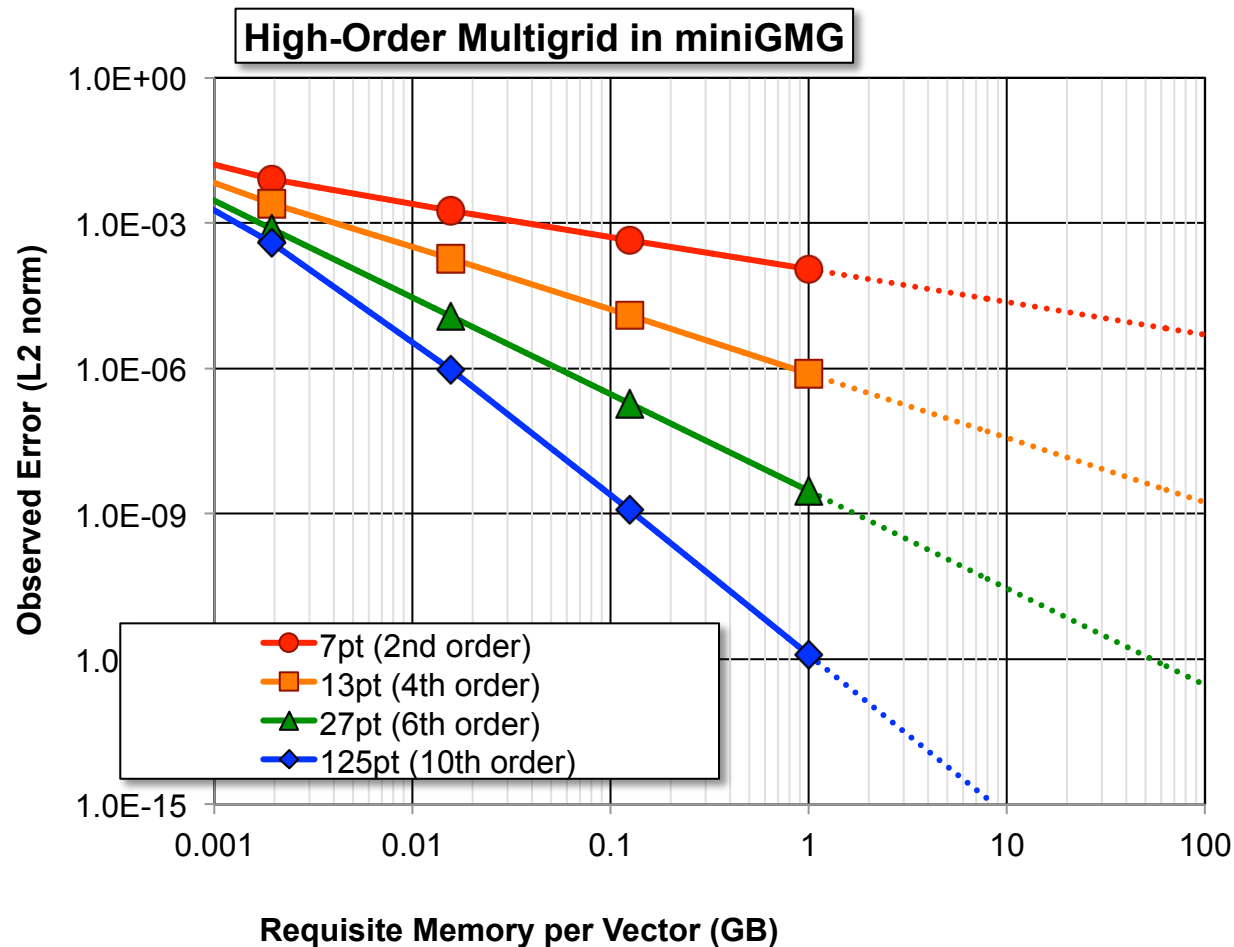
Performance on K20c

% DRAM Bandwidth achieved by GSRB
Smooth on 64, 64³ boxes



2D Thread Blocks <TX, TY>

Higher-Order Stencils



Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse

Higher-Order Stencils

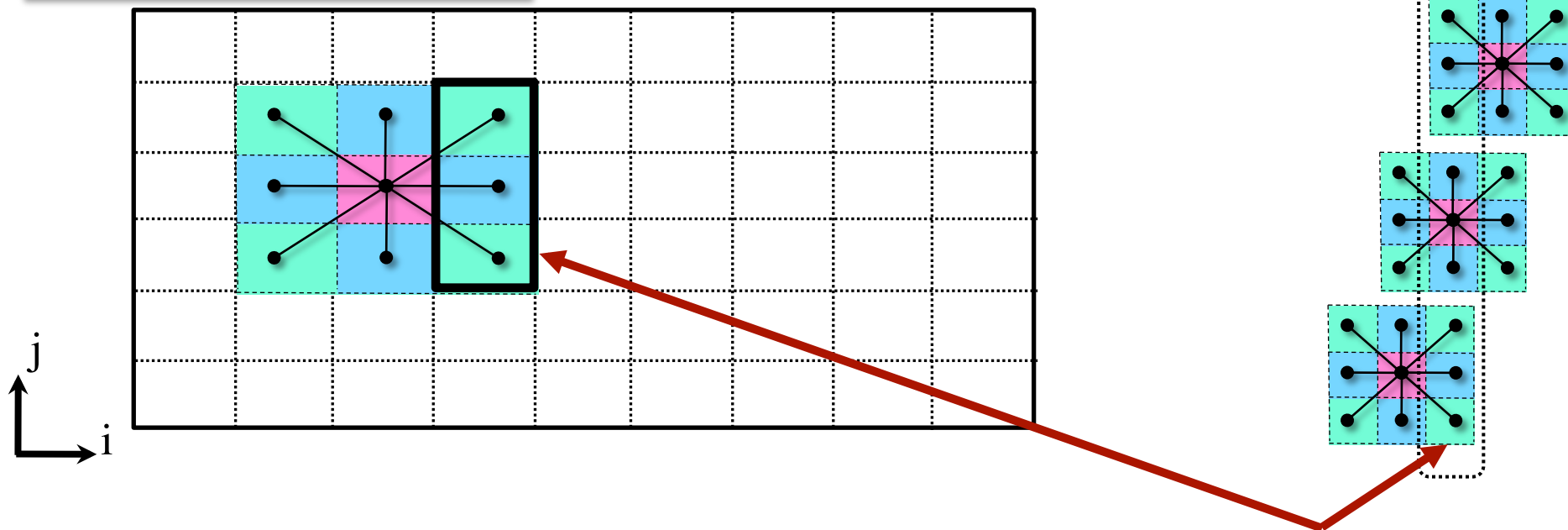
Stenci	Coefficien	Iteration	Flop	Byte	AI
7-point	Constant	Jacobi	8	24	0.33
13-point	Constant	Jacobi	15	24	0.63
27-point	Constant	Jacobi	32	24	1.33
125-point	Constant	Jacobi	134	24	5.58



Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse

Partial Sums

2D 9-point CC stencil



```
for (j=0; j<N; j++)  
  for (i=0; i<N; i++){  
    out[k][j][i] = w1*  
      (in[j-1][i ] + in[j+1][i]  
      + in[j ][i-1] + in[j ][i+1] )  
    + w2 *(in[j-1][i-1] + in[j+1][i-1]  
    + in[j-1][i+1] + in[j+1][i+1] )  
    + w3* in[j ][i ];  
  }
```

Right (leading) edge of points from the input grid is reused in the next two iterations of the inner-loop

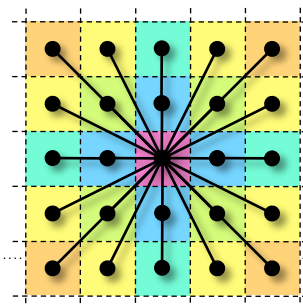
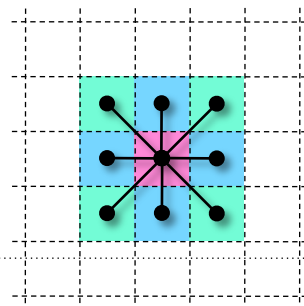
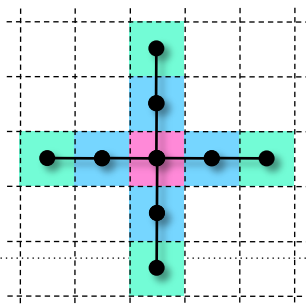
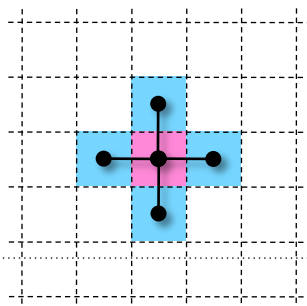
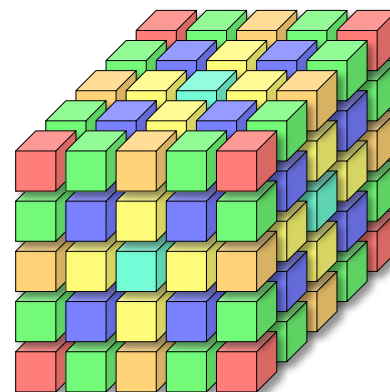
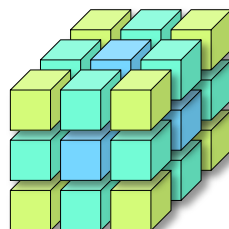
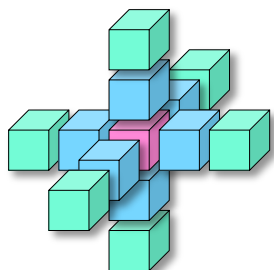
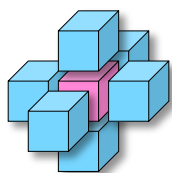
The right edge acts as the center and left edge for the next iterations

Partial Sums

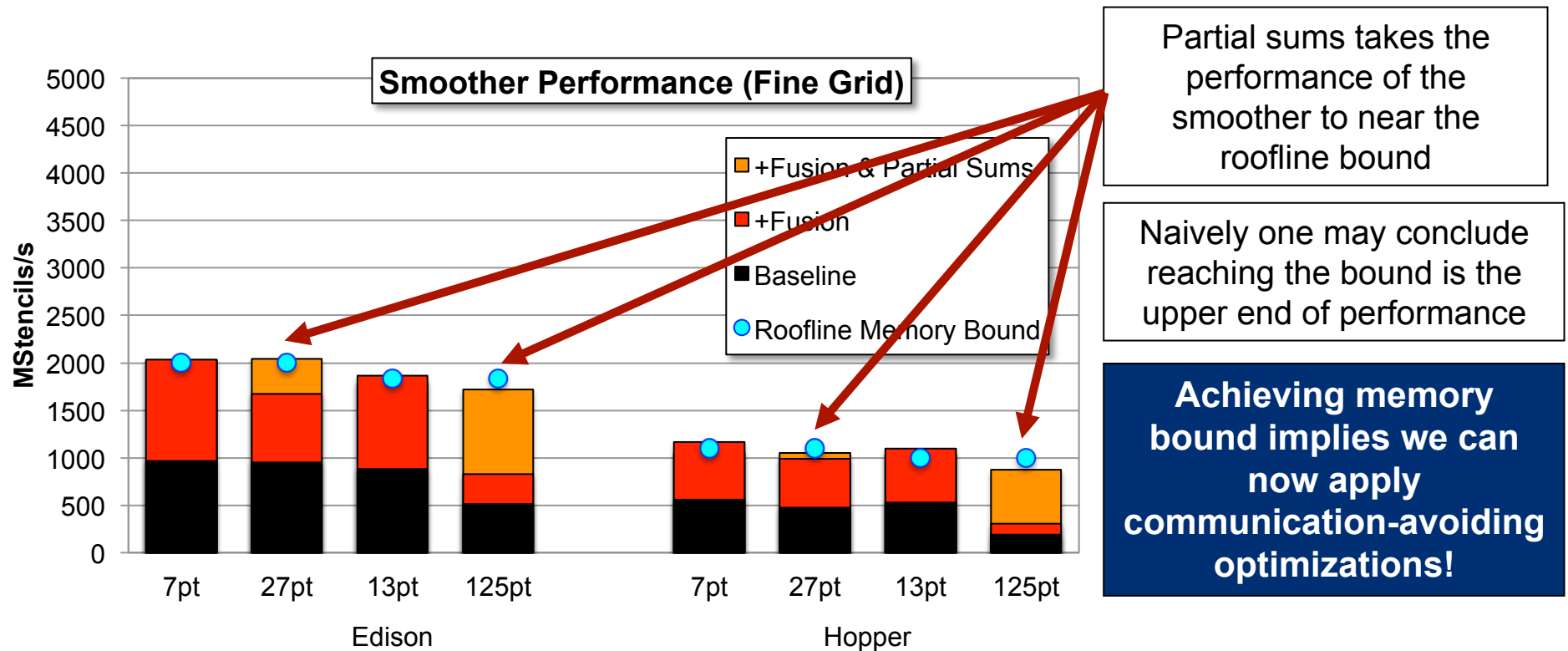
For 3D stencils we pick the leading plane

Exploiting symmetry reduces flops significantly for 27, 125-pt stencils

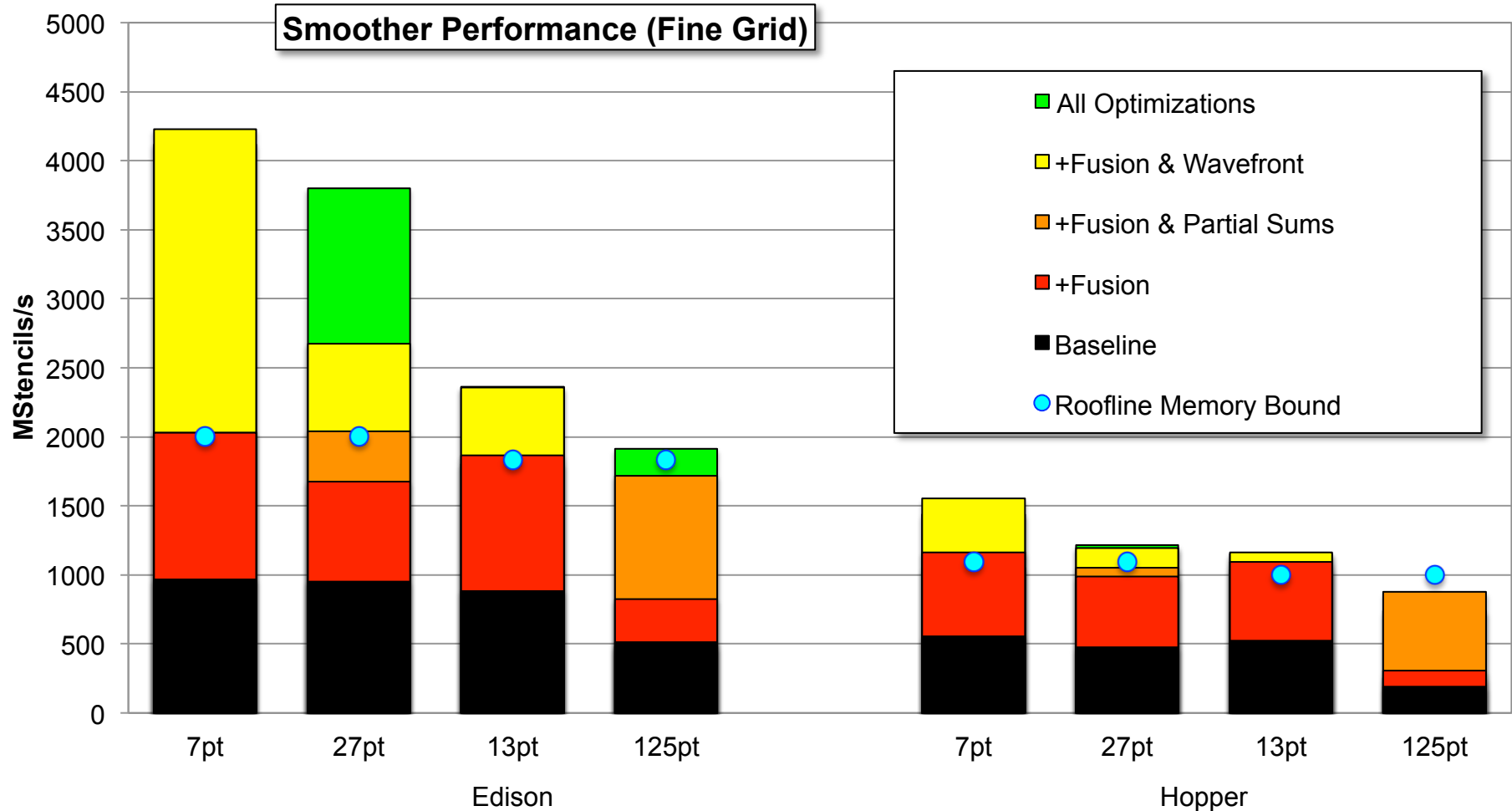
For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction)



Smooth Performance



Smooth Performance



Transformation must work with other CA optimizations!

Partial Sums – CHiLL Script

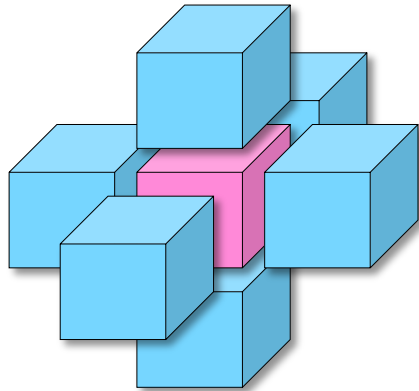
```
/* jacobi_box_4_64.py, 27-pt stencil, 643 box size */  
from chill import *  
  
#select which computation to optimize  
source('jacobi_box_4_64.c')  
procedure('smooth_box_4_64')  
loop(0)  
original() # fuse wherever possible  
  
#create a parallel wavefront  
skew([0,1,2,3,4,5],2,[2,1])  
permute([2,1,3,4])  
  
#partial sum for high order stencils and fuse result  
distribute([0,1,2,3,4,5],2)  
stencil_temp(0)  
stencil_temp(5)  
fuse([2,3,4,5,6,7,8,9],1)  
fuse([2,3,4,5,6,7,8,9],2)  
fuse([2,3,4,5,6,7,8,9],3)  
fuse([2,3,4,5,6,7,8,9],4)
```

Summary and Conclusions

- Compiler technology can be leveraged for automated architecture-specific optimization from high-level specification for several motifs
- Compiler technology allows composing a sequence of transformations, and mixing known and novel domain-specific optimizations
- Performance rivaling manually-tuned code and sometimes better

Extra Slides

Arithmetic Intensity (AI) of Stencil Computation



```
for (k=0; k<N; k++)  
  for (j=0; j<N; j++)  
    for (i=0; i<N; i++){
```

```
      phi_out[k][j][i] = w1 * phi_in[k][j][i]  
        + w2 * (phi_in[k+1][j][i] + phi_in[k-1][j][i]  
        + phi_in[k][j+1][i] + phi_in[k][j-1][i]  
        + phi_in[k][j+1][i] + phi_in[k][j-1][i]);  
    }
```

6 adds+2 muls = 8
flops

Read N^3 Grid (phi_in)

Write Allocate N^3 Grid (phi_out)

Write N^3 Grid (phi_out)

Ideal cache behavior, compulsory (cold) misses only

$$\frac{\text{Floating Point Ops (flops)}}{\text{Data Moved (Bytes)}} = \frac{8 * N^3}{3 * N^3 * 8} = 0.33$$

AI

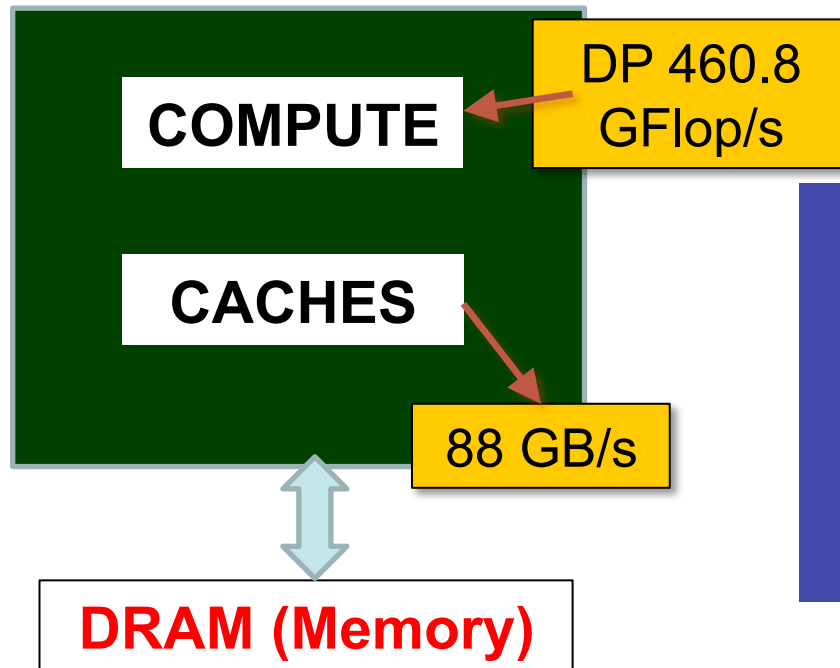
Machine Balance (Edison)



(very) Abstract Node

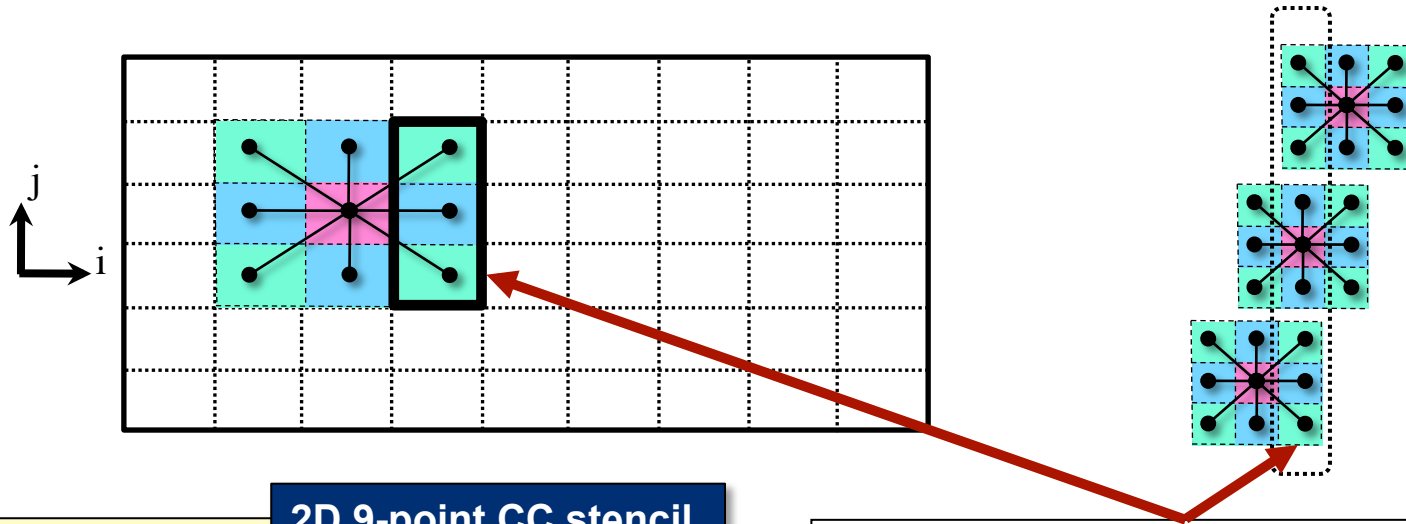
$$\frac{\text{Floating Point Ops per second}}{\text{DRAM Memory Bandwidth}} = \frac{460.8}{88}$$

Machine Balance 5.2



To achieve peak performance, code needs to execute 5.2 flops per byte!

Opportunity for Data and Computation Reuse



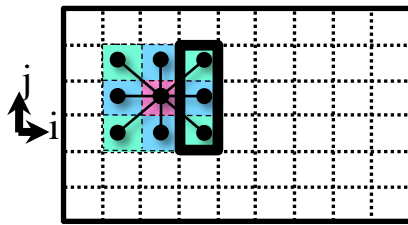
2D 9-point CC stencil

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++){
    out[k][j][i] = w1*
      (in[j-1][i] + in[j+1][i]
       + in[j][i-1] + in[j][i+1])
    + w2 *(in[j-1][i-1] + in[j+1][i-1]
           + in[j-1][i+1] + in[j+1][i+1])
    + w3* in[j][i];
  }
```

Right (leading) edge of points from the input grid is reused in the next two iterations of the inner-loop

The right edge acts as the center and left edge for the next iterations

Buffering Partial Sums: Exploiting Reuse



Partial Sum B0

Right Edge for

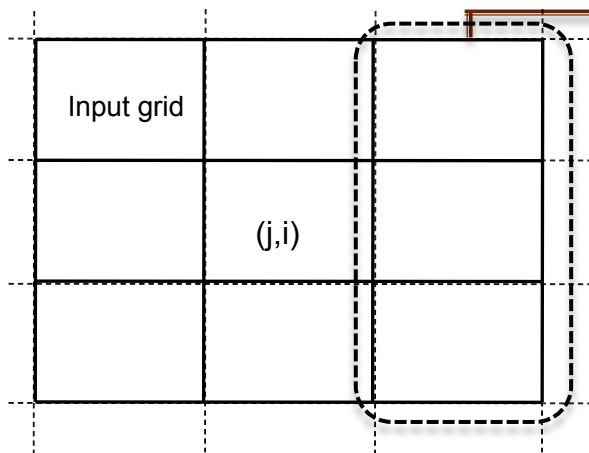
Partial Sum B1

Center for (j,i+1)

Partial Sum B2

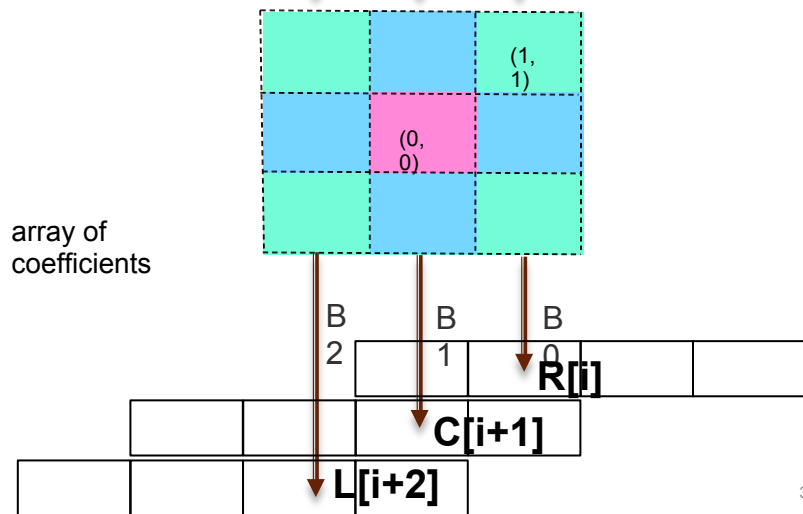
Left Edge for (j,i+2)

Partial Sums are buffered in linear buffers R , C, L

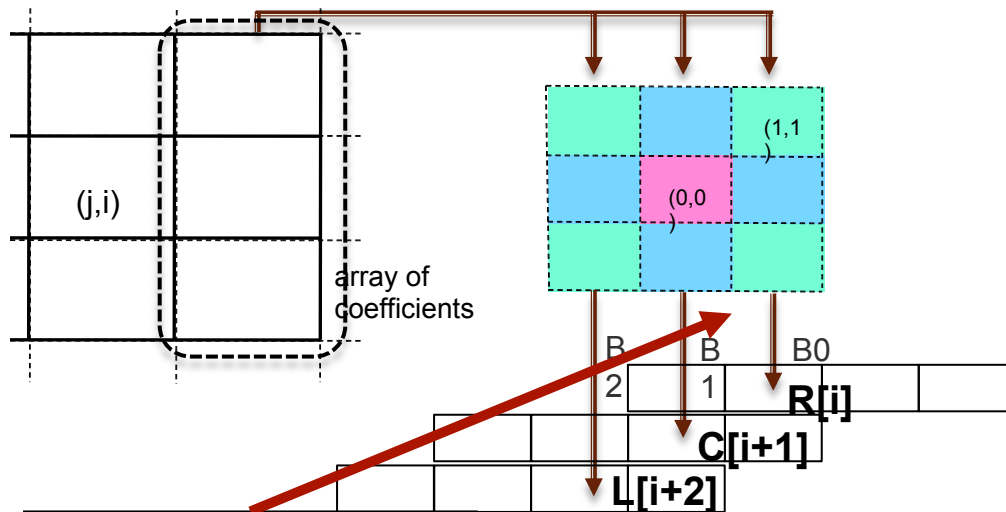


$$\text{out } [j][i] = R [i] + C [i] + L [i]$$

array of coefficients



Exploiting Symmetry to Reduce Computation

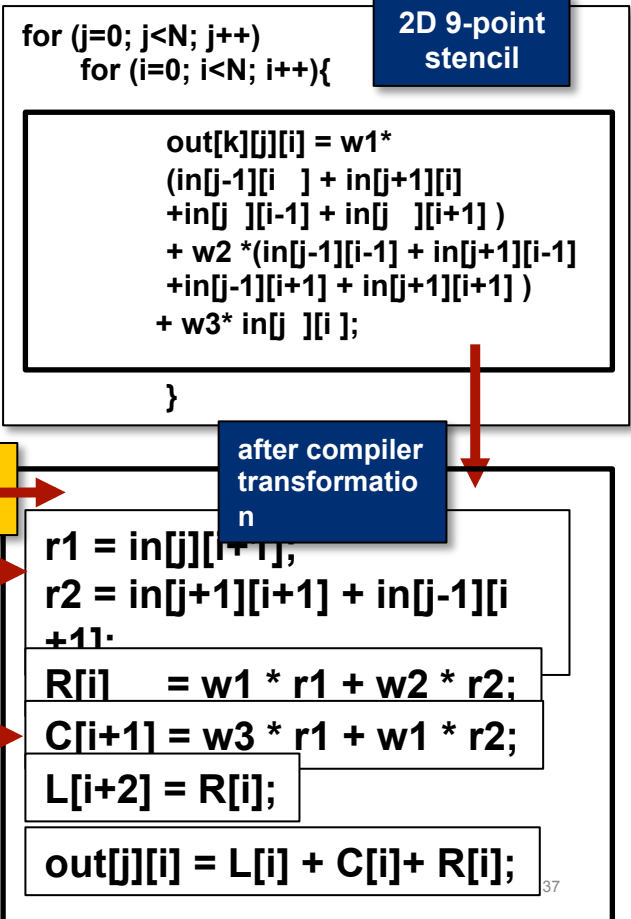


Due to symmetry of coefficients the loaded edge (plane) can be factored into sums

factor the points multiplied by the same coefficient

Factors are reused to compute partial sums saving flops (5 adds instead of 8)

loads the right edge of points

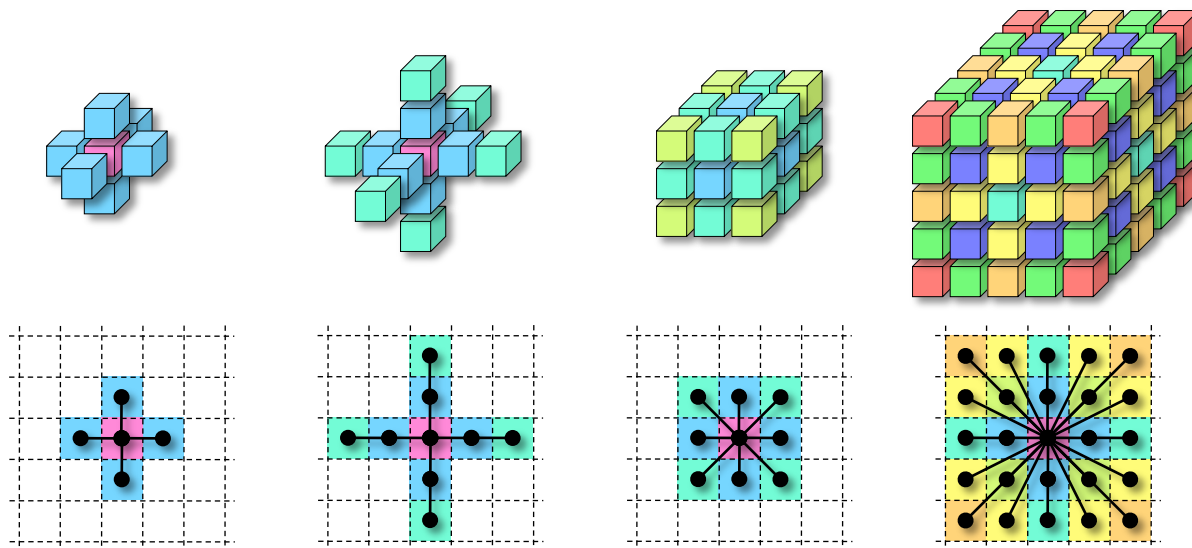


Exploiting Symmetry to Reduce Computation

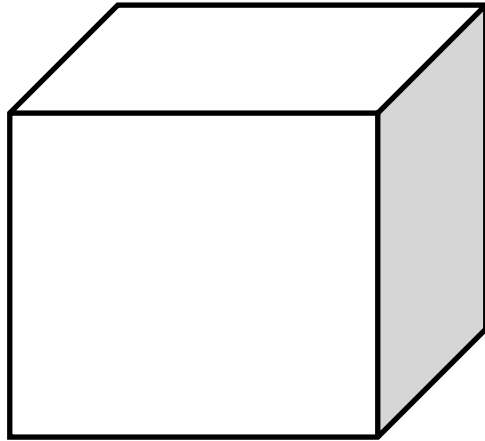
For 3D stencils we pick the leading plane

Exploiting symmetry reduces flops significantly for 27, 125-pt stencils

For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction)

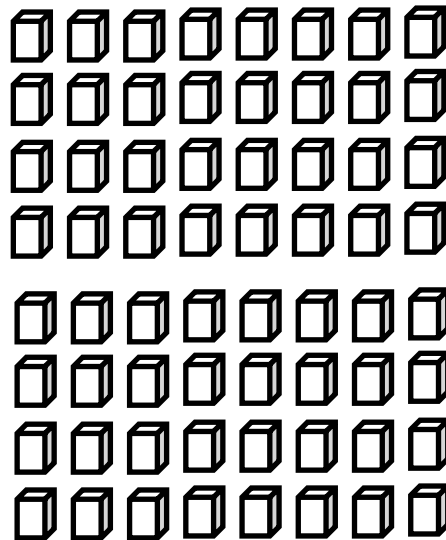


Domain 256^3

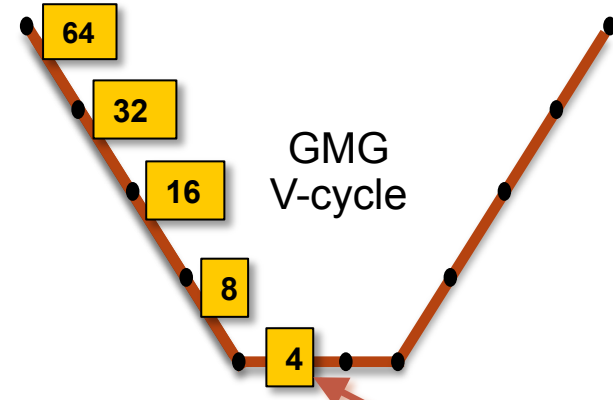


miniGMG

List of 64^3 Boxes
Computed In Parallel (OMP)

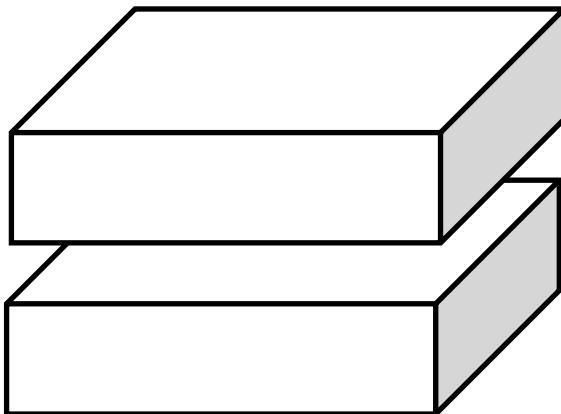


4 iterations
of smooth



48 iterations
of Smooth

Domain
decomposed to MPI
processes (2)



**Smooth Dominates
Runtime**